Lecture 16
Decompilation

Why decompilation?

This course is ostensibly about Optimising Compilers.

It is really about *program analysis and transformation*.

Decompilation is achieved through analysis and transformation of target code; the transformations just work in the opposite direction.

The decompilation problem

Even simple compilation discards a lot of information:

- Comments
- Function and variable names
- Structured control flow
- Type information

The decompilation problem

Optimising compilation is even worse:

- Dead code and common subexpressions are eliminated
- Algebraic expressions are rewritten
- Code and data are inlined; loops are unrolled
- Unrelated local variables are allocated to the same architectural register
- Instructions are reordered by code motion optimisations and instruction scheduling

The decompilation problem

Some of this information is never going to be automatically recoverable (e.g. comments, variable names); some of it we may be able to partially recover if our techniques are sophisticated enough.

Compilation is *not injective*. Many different source programs may result in the same compiled code, so the best we can do is to pick a reasonable *representative* source program.

Intermediate code

It is relatively straightforward to extract a flowgraph from an assembler program.

Basic blocks are located in the same way as during forward compilation; we must simply deal with the semantics of the target instructions rather than our intermediate 3-address code.

Intermediate code

For many purposes (e.g. simplicity, retargetability) it might be beneficial to convert the target instructions back into 3-address code when storing it into the flowgraph.

This presents its own problems: for example, many architectures include instructions which test or set condition flags in a status register, so it may be necessary to laboriously reconstruct this behaviour with extra virtual registers and then use dead-code elimination to remove all unnecessary instructions thus generated.

Control reconstruction

A compiler apparently destroys the high-level control structure which is evident in a program's source code.

After building a flowgraph during decompilation, we can recover some of this structure by attempting to match *intervals* of the flowgraph against some fixed set of familiar syntactic forms from our high-level language.

Finding loops

Any structured loops from the original program will have been compiled into tests and branches; they will look like arbitrary (“spaghetti”) control flow.

In order to recover the high-level structure of these loops, we must use *dominance*.

Dominance

In a flowgraph, we say a node *m* *dominates* another node *n* if control must go through *m* before it can reach *n*.

A node *m* *strictly dominates* another node *n* if *m* dominates *n* and $m \neq n$.

The *immediate dominator* of a node *n* is the unique node that strictly dominates *n* but doesn't dominate any other strict dominator of *n*.

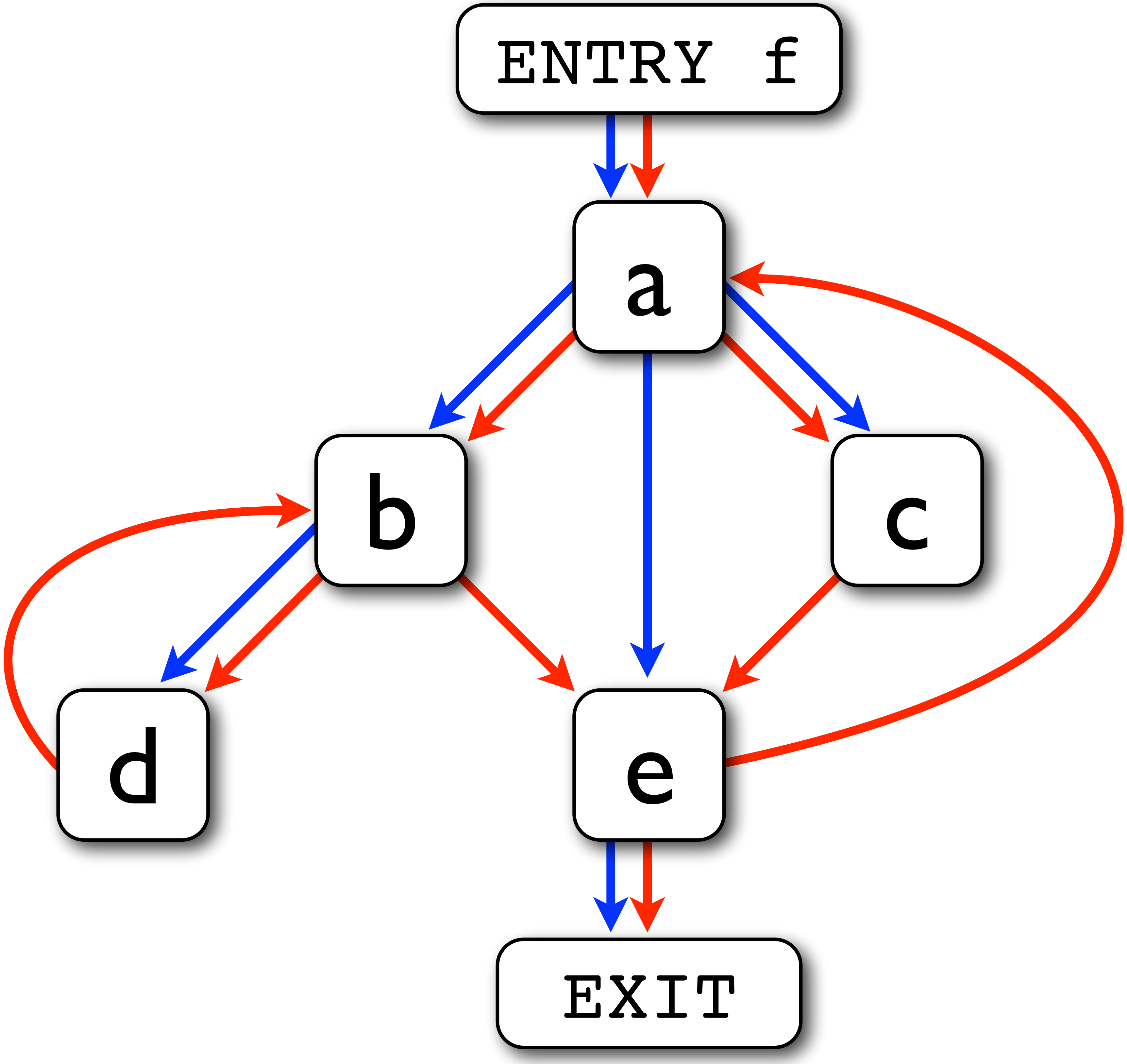
Dominance

A node *n* is in the *dominance frontier* of a node *m* if *m* does not strictly dominate *n* but does dominate an immediate predecessor of *n*.

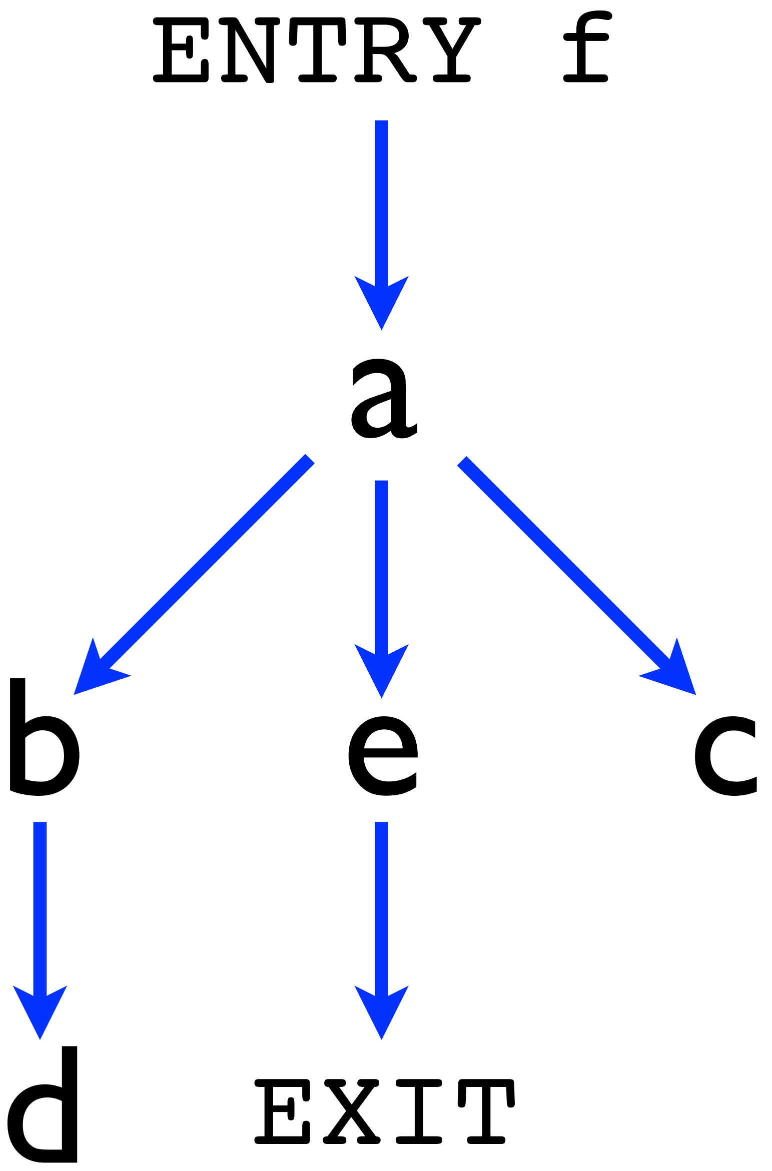
Intuitively this is the set of nodes where *m*'s dominance stops.

We can represent this dominance relation with a *dominance tree* in which each edge connects a node with its immediate dominator.

Dominance



Dominance



ENTRY f

a

b

e

c

d

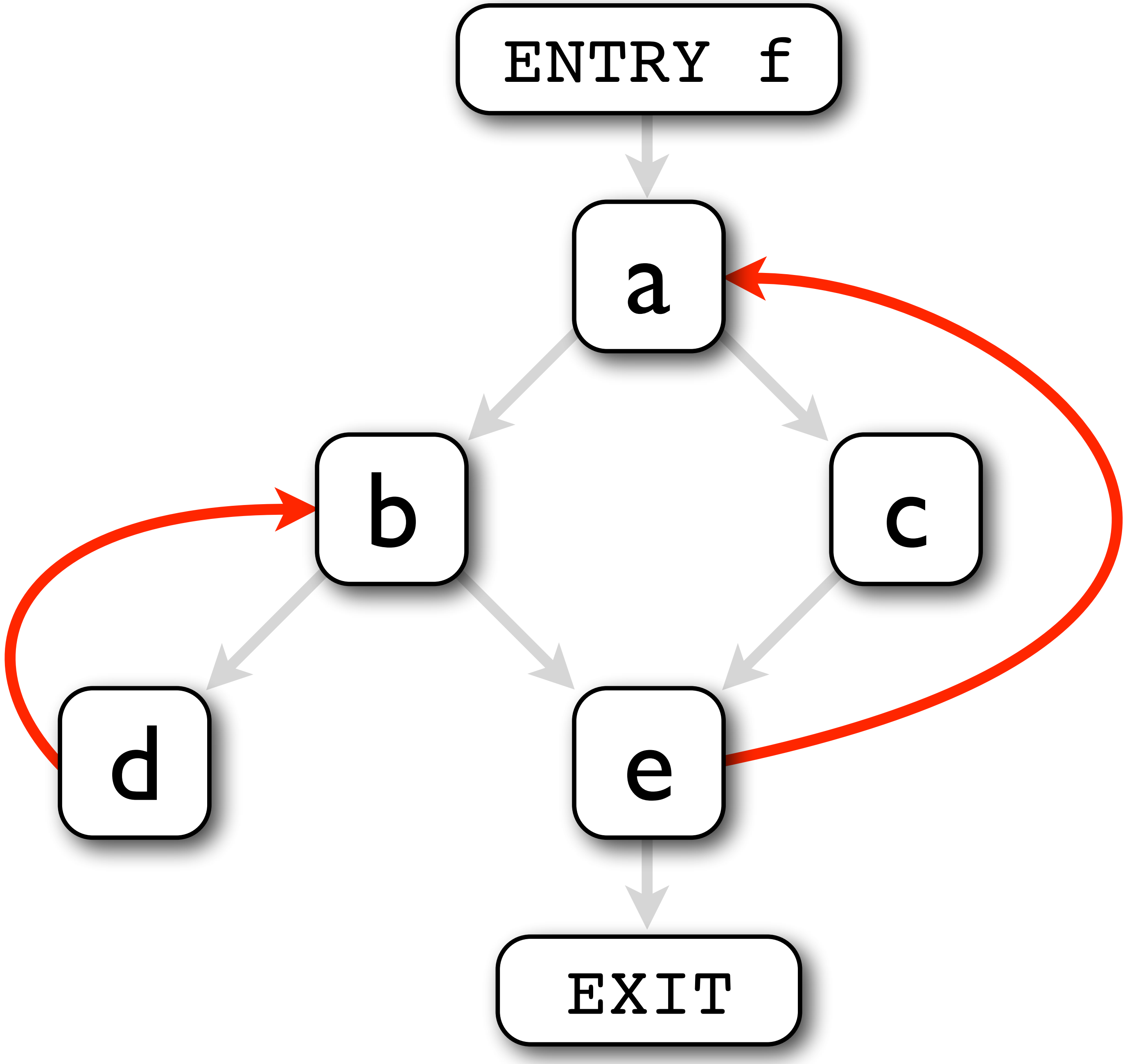
EXIT

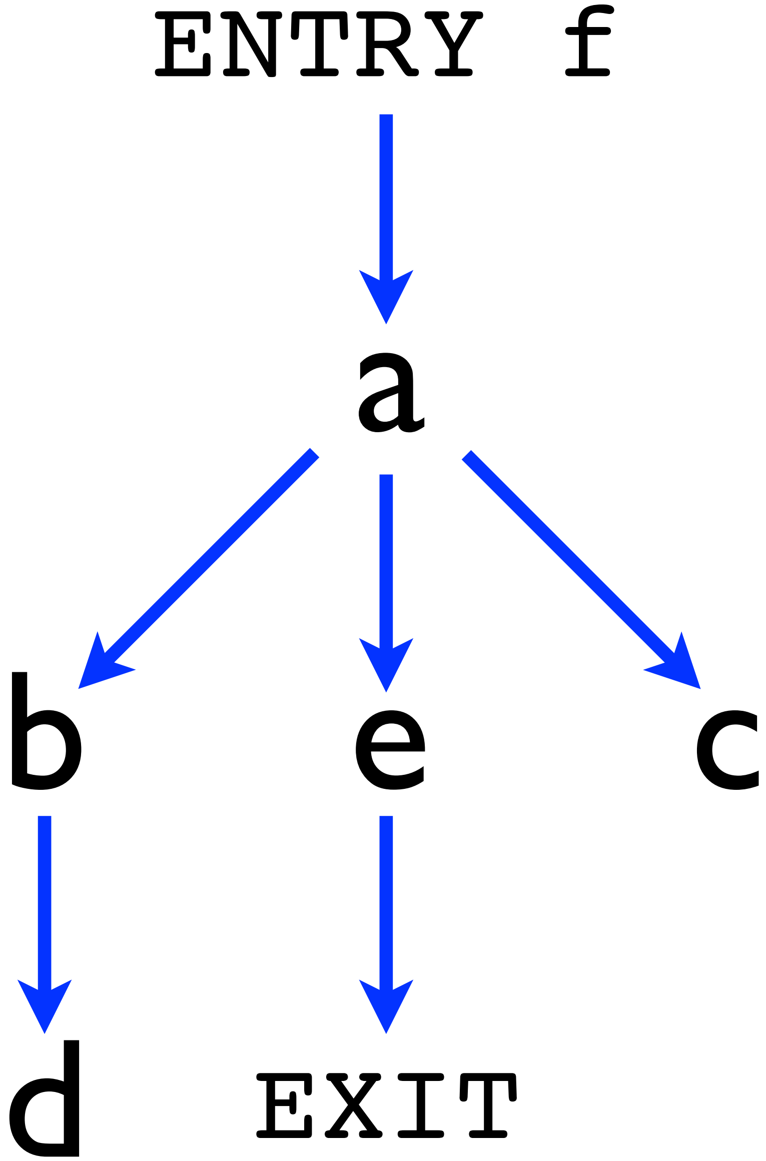
Back edges

We can now define the concept of a *back edge*.

In a flowgraph, a back edge is one whose head dominates its tail.

Back edges



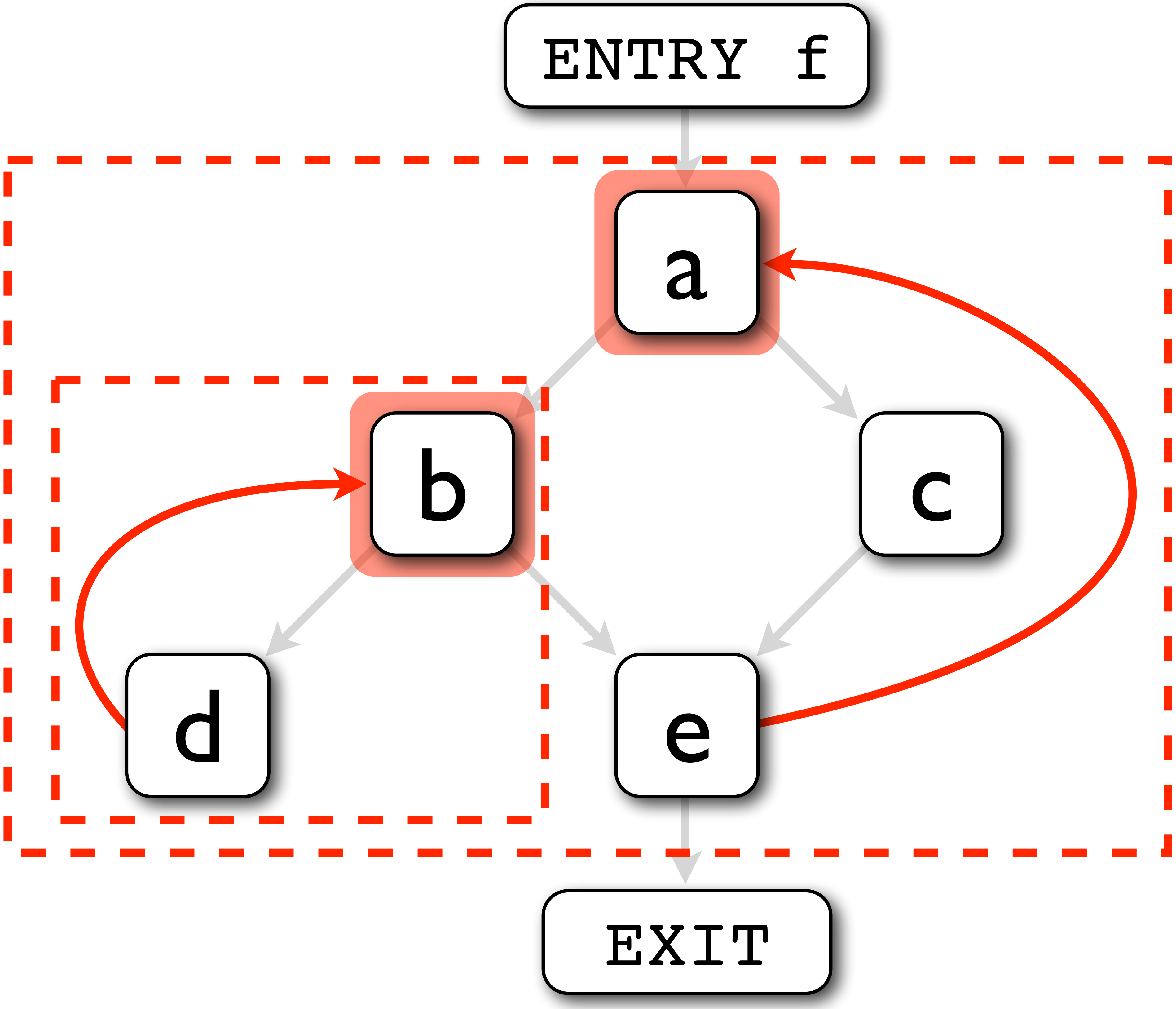


Finding loops

Each back edge has an associated loop.

The head of a back edge points to the *loop header*, and the *loop body* consists of all the nodes from which the tail of the back edge can be reached without passing through the loop header.

Finding loops

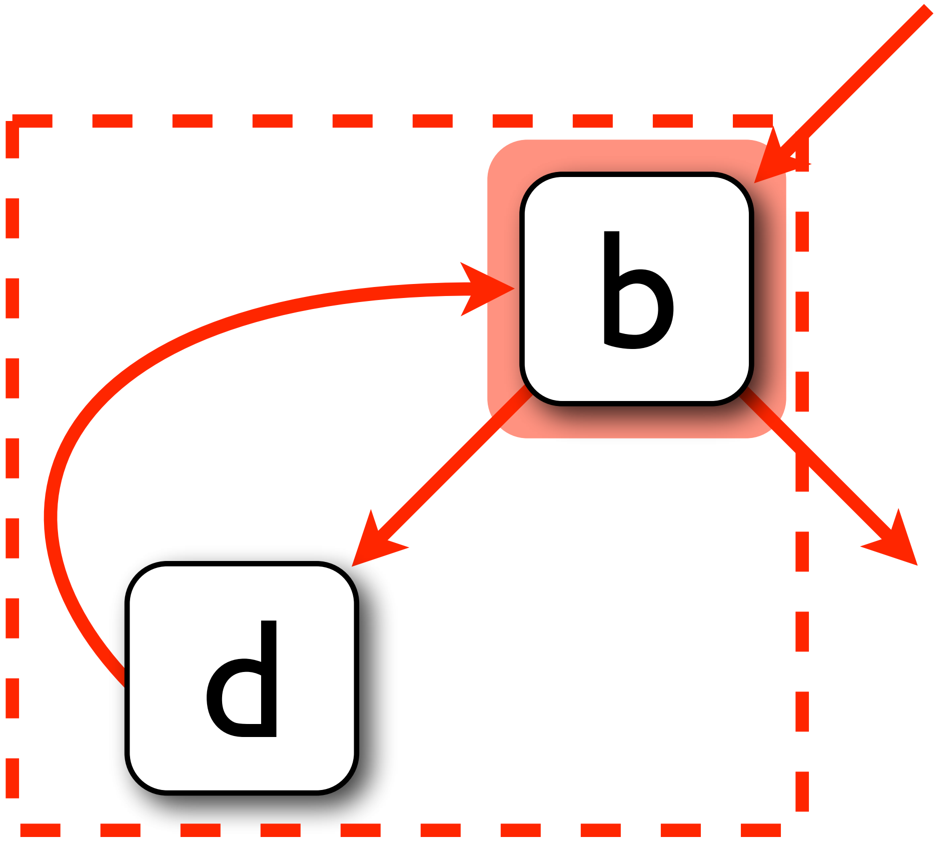


Finding loops

Once each loop has been identified, we can examine its structure to determine what kind of loop it is, and hence how best to represent it in source code.

Finding loops

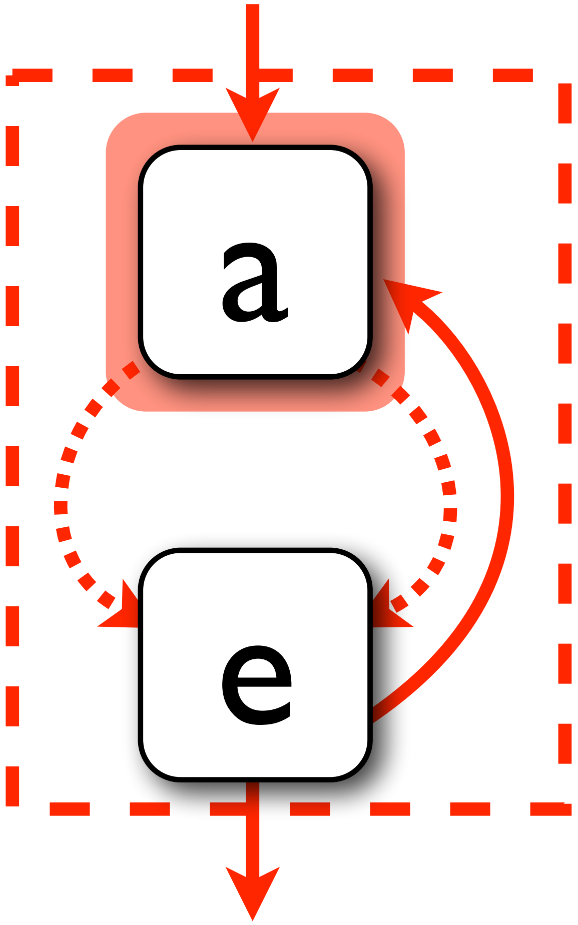
Here, the loop header contains a conditional which determines whether the loop body is executed, and the last node of the body unconditionally transfers control back to the header.



This structure corresponds to source-level `while (...) {...}` syntax.

Finding loops

Here, the loop header unconditionally allows the body to execute, and the last node of the body tests whether the loop should execute again.

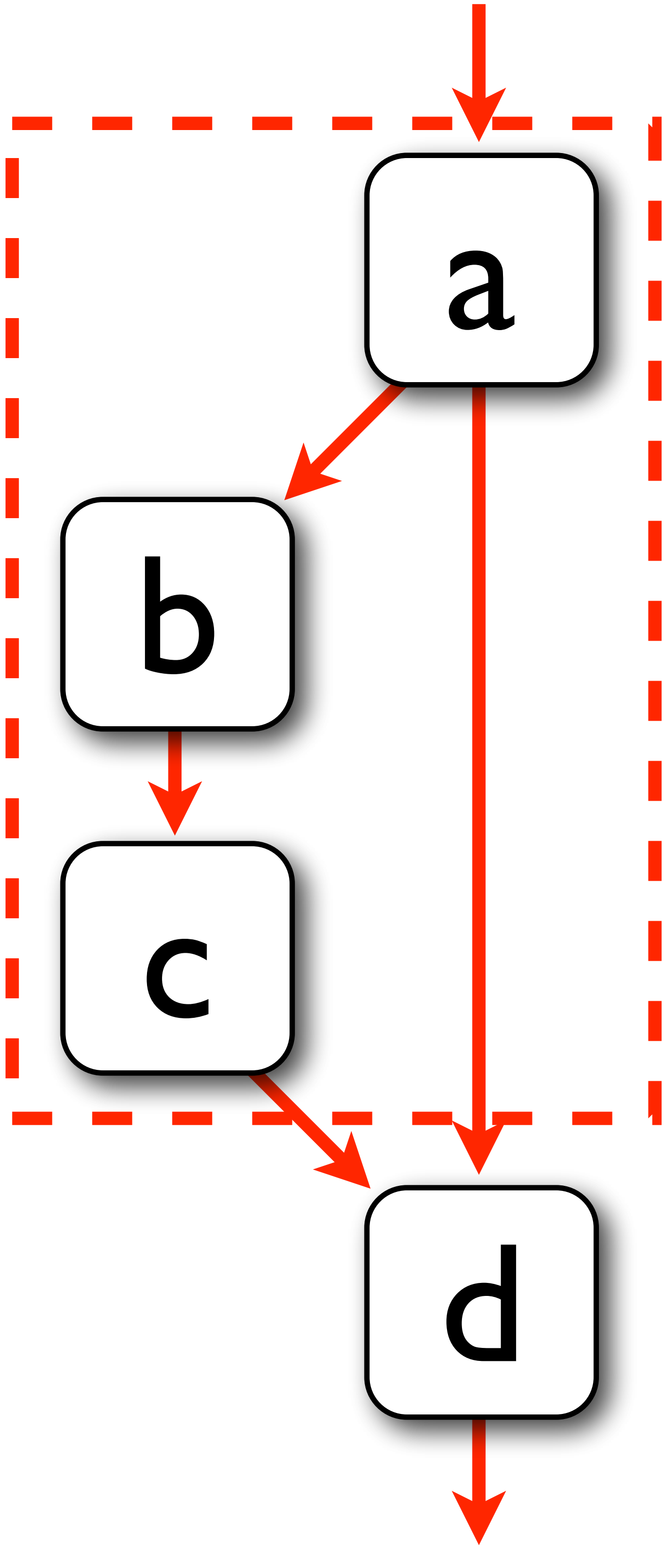


This structure corresponds to source-level `do {...} while (...)` syntax.

Finding conditionals

A similar principle applies when trying to reconstruct conditionals: we look for structures in the flowgraph which may be represented by particular forms of high-level language syntax.

Finding conditionals

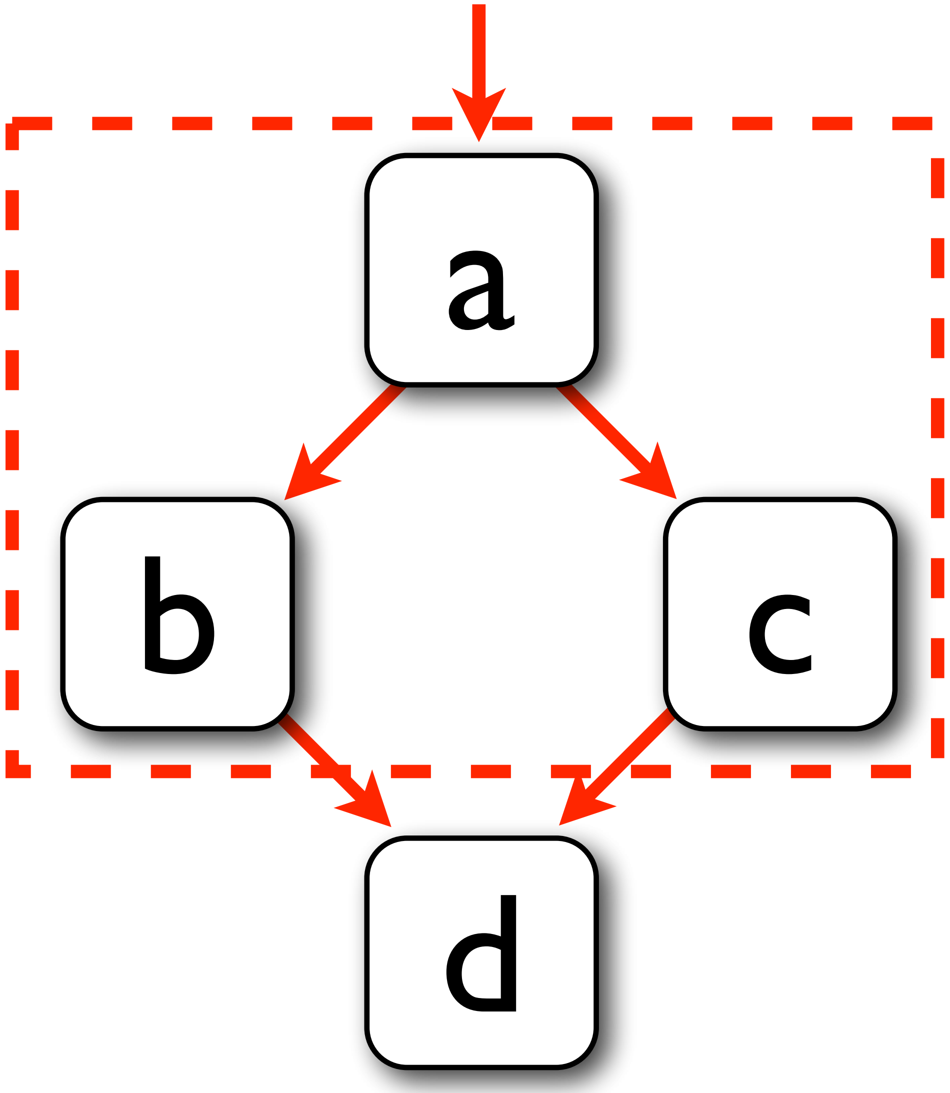


The first node in this interval transfers control to one node if some condition is true, otherwise it transfers control to another node (which control also eventually reaches along the first branch).

This structure corresponds to source-level `if (...) then {...}` syntax.

Finding conditionals

The first node in this interval transfers control to one node if some condition is true, and another node if the condition is false; control always reaches some later node.



This structure corresponds to source-level

```
if (...) then {...}
else {...}
```

syntax.

Control reconstruction

We can keep doing this for whatever other control-flow constructs are available in our source language.

Once an interval of the flowgraph has been matched against a higher-level control structure in this way, its entire subgraph can be replaced with a single node which represents that structure and contains all of the information necessary to generate the appropriate source code.

Type reconstruction

Many source languages also contain rich information about the *types* of variables: integers, booleans, arrays, pointers, and more elaborate data-structure types such as `unions` and `structs`.

At the target code level there are no variables, only registers and memory locations.

Types barely exist here: memory contains arbitrary bytes, and registers contain integers of various bit-widths (possibly floating-point values too).

Type reconstruction

Reconstruction of the types of source-level variables is made more difficult by the combination of SSA and register allocation performed by an optimising compiler.

SSA splits one user variable into many variables — one for each static assignment — and any of these variables with disjoint live ranges may be allocated to the same architectural register.

Type reconstruction

So each user variable may be spread between several registers — and each register may hold the value of different variables at different times.

It’s therefore a bit hopeless to try to give a type to each architectural register; the notional type of the value held by any given register will change during execution.

```
int x = 42;                MOV r3,#42
⋮                 →        ⋮
char *y = “42”;            MOV r3,#0xFF34
```

Type reconstruction

Happily, we can undo the damage by once again converting to SSA form: this will split a single register into many registers, each of which can be assigned a different type if necessary.

```
MOV r3,#42
⋮
MOV r3,#0xFF34
```

→

```
MOV r3_a,#42
⋮
MOV r3_b,#0xFF34
```

Type reconstruction

C

```
int foo (int *x) {
  return x[1] + 2;
}
```

↓ compile

ARM

```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

C

```
int f (int r0) {
  r0 = *(int *)(r0 + 4);
  r0 = r0 + 2;
  return r0;
}
```

decompile

ARM

```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

```
int f (int r0) {
  r0 = *(int *)(r0 + 4);
  r0 = r0 + 2;
  return r0;
}
```

↓ SSA

```
int f (int r0_a) {
  int r0_b = *(int *)(r0_a + 4);
  int r0_c = r0_b + 2;
  return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
   int r0_b = *(r0_a + 1);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

reconstruct types

```
int f (int r0_a) {
   int r0_b = *(int *)(r0_a + 4);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
   int r0_b = *(r0_a + 1);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

↓ reconstruct syntax

```
int f (int *r0_a) {
   int r0_b = r0_a[1];
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
   int f (int *r0_a) {
     return r0_a[1] + 2;
   }
```

propagate copies

```
int f (int *r0_a) {
  int r0_b = r0_a[1];
  int r0_c = r0_b + 2;
  return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
  return r0_a[1] + 2;
}
```

In fact, the return type could be anything, so more generally:

```
T f (T *r0_a) {
  return r0_a[1] + 2;
}
```

Type reconstruction

This is all achieved using constraint-based analysis: each target instruction generates constraints on the types of the registers, and we then solve these constraints in order to assign types at the source level.

Typing information is often incomplete intraprocedurally (as in the example); constraints generated at call sites help to fill in the gaps.

We can also infer `unions`, `structs`, etc.

Summary

- Decompilation is another application of program analysis and transformation
- Compilation discards lots of information about programs, some of which can be recovered
- Loops can be identified by using dominator trees
- Other control structure can also be recovered
- Types can be partially reconstructed with constraint-based analysis